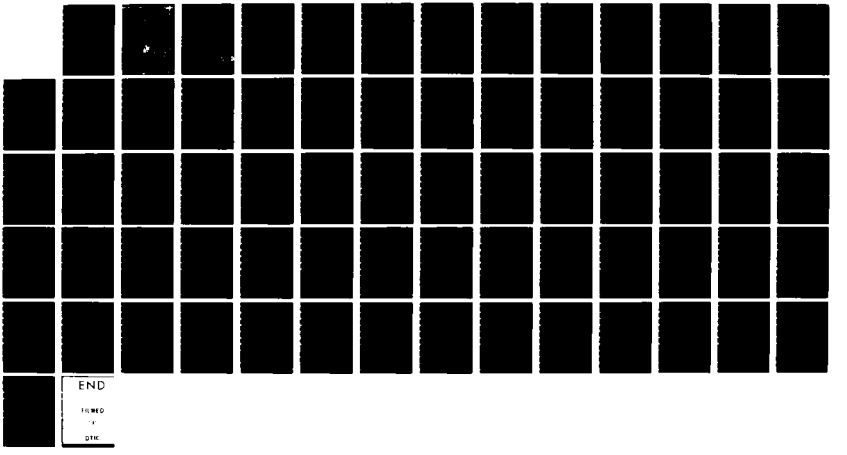
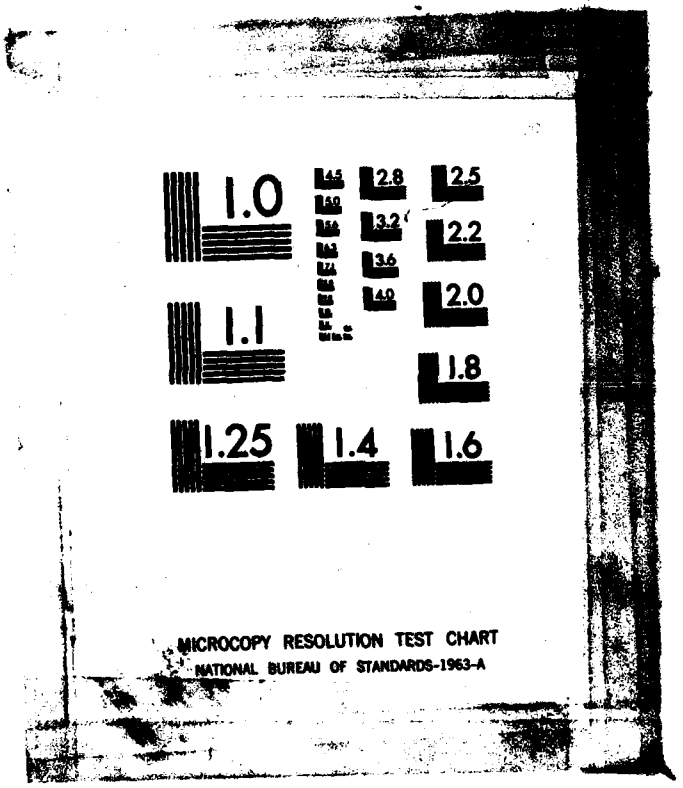
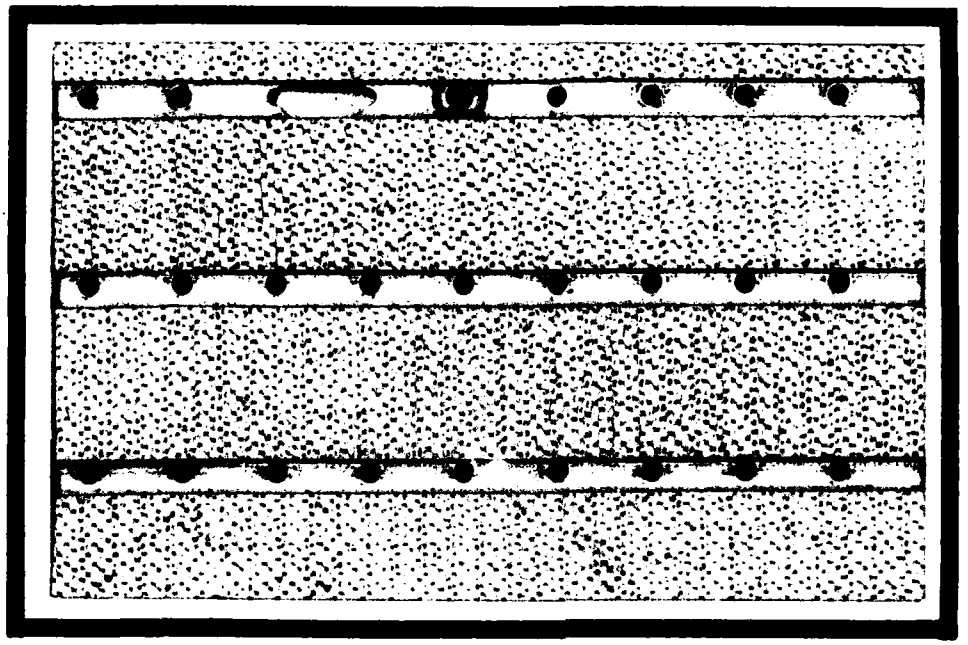
AD-A124 765 VALIDATION IN ADA PROGRAMMING SUPPORT ENVIRONMENTS(U) 1/1
VIRGINIA POLYTECHNIC INST AND STATE UNIV BLACKSBURG
COMPUTER S.. D KAFURA ET AL. DEC 82 CSIE-82-12
UNCLASSIFIED N00014-81-K-0143 F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



Virginia Polytechnic Institute and State University

Computer Science
Industrial Engineering and Operations Research
BLACKSBURG, VIRGINIA 24061

83 02 022 027

CSIE-82-12

December 1982

VALIDATION IN ADA PROGRAMMING SUPPORT ENVIRONMENTS

Dennis Kafura, J.A.N. Lee, Timothy Lindquist
Virginia Tech, Blacksburg, VA

and

Thomas Probert
MITRE Corp., McLean, VA

TECHNICAL REPORT

Prepared for
Engineering Psychology Group, Office of Naval Research
ONR Contract Number N00014-81-K-0143
Work Unit Number NR SRO-101

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CSIE-82-12	2. GOVT ACCESSION NO. AD-A134 765	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) VALIDATION IN ADA PROGRAMMING SUPPORT ENVIRONMENTS		5. TYPE OF REPORT & PERIOD COVERED Technical
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Dennis Kafura, J. A. N. Lee, Timothy Lindquist and Thomas Probert		8. CONTRACT OR GRANT NUMBER(s) N00014-81-K-0143
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Virginia Polytechnic Institute & State University Blacksburg, VA 24061		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61153N42; RR04209; RR0420901; NR SRO-101
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research, Code 442 800 North Quincy Street Arlington, VA 22217		12. REPORT DATE December 1982
		13. NUMBER OF PAGES 67
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Ada, Ada Programming Support Environments, Programming Environment, Validation, Standards, Interfaces

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

To this date validation has been applied in only two areas, in the validation of programs and the validation of compilers, and then not to any degree which can truly be classified as more than "empirical". This study was established to investigate the steps which would be needed to extend those previous experiences into the realm of programming environments and in particular the environments being proposed for use in the Ada program. A model of such environments already exists but is found to be lacking in essential detail necessary for an implementation to prescribe a model by which validation

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (continued)

can be specified. This report does not itself provide any details of specific validation procedures or mechanisms, but rather investigates the processes for Ada Programming Support Environment (APSE) implementation in terms of the Ada Programming Language, and uses those specifications to suggest a mechanism for validation suite development.

Further in order to accomplish these goals it is suggested that the conceptual model of the "STONEMAN" document be extended to express the wider computing environments in which the APSE would reside. This extended model would also provide a fundamental basis for the design of Ada systems which responds to the need to provide networking, distributed processing and security enclaves.

DEPARTMENT OF COMPUTER SCIENCE
VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY
BLACKSBURG VA 24061

VALIDATION IN ADA* PROGRAMMING SUPPORT ENVIRONMENTS**

by
Dennis Kafura, J.A.N. Lee, Timothy Lindquist
Virginia Tech, Blacksburg VA
and
Thomas Probert
MITRE Corp., McLean VA

WORKING PAPER
[Printed 83/01/07]

Abstract

To this date validation has been applied in only two areas, in the validation of programs and the validation of compilers, and then not to any degree which can truly be classified as more than "empirical". This study was established to investigate the steps which would be needed to extend those previous experiences into the realm of programming environments and in particular the environments being proposed for use in the Ada program. A model of such environments already exists but is found to be lacking in essential detail necessary for an implementation to prescribe a model by which validation can be specified. This report does not itself provide any details of specific validation procedures or mechanisms, but rather investigates the processes for Ada Programming Support Environment (APSE) implementation in terms of the Ada Programming Language, and uses those specifications to suggest a mechanism for validation suite development.

Further in order to accomplish these goals it is suggested that the conceptual model of the "STONEMAN" document be extended to express the wider computing environments in which the APSE would reside. This extended model would also provides a fundamental basis for the design of Ada systems which respond to the need to provide networking, distributed processing and security enclaves.

---

* Ada is a registered Trademark of the Ada Joint Program Office of the U.S. Dept. of Defense.

** This research was supported by the Ada Joint Program Office, U.S. Department of Defense, through the Office of Naval Research under contract number N00014-81-K-0143 and work unit number NR-SRO-101. The effort was supported by the Engineering Psychology Group, Office of Naval Research under the Technical Direction of Dr. John J. O'Hare. Reproduction in whole or in part is permitted for any purpose of the United States Government.

CONTENTS

INTRODUCTION *

A fundamental objective of the Department of Defense (DoD) initiative to develop Ada is to increase the portability and maintainability of embedded software [1]. The Ada language will be the common high order language for use in future DoD embedded systems, and a Ada Validation Organization (AVO) has been established to ensure that Ada compilers implement the same common language. A major objective of the Ada Joint Program Office (AJPO) is to ensure that Ada remains as independent of computing systems and applications as possible, and has undertaken a standardization process to accomplish this objective. The Ada language is a Military Standard (MIL STD 1815) and has been proposed as both an American National (ANSI) and International (ISO) Standard. Requirements for a common (standardized) Ada Programming Support Environment (APSE) have been defined but the details have not yet been settled; however there is a growing realization that some form of the APSE or its kernel computing system interface (KAPSE) may eventually be standardized and conforming products be subjected to validation.

Although validation has usually been an afterthought in language design and implementation, this is not the case for Ada. At least in the case of the programming language the development of the validation suite of test programs has been accomplished hand-in-hand with the language standardization process and the ongoing

---

* This introduction is based on: Probert, T.H., Ada Validation Organization: Policies and Procedures, Rep. No. MTR-82W00103, The MITRE Corp., McLean VA, 1982 June, 27pp.

implementation activities. This report surveys the problem of developing methods and techniques for the validation of APSEs based on the preliminary design requirements available at this time. To date, validation activities within the U.S. Federal Government have been restricted to implementations of programming languages according to Federal procurement requirements and independent of the needs of the general industry. In some ways this activity fulfilled that need and at the same time has had the side-effect of providing a measure of "quality assurance" to the non-government consumers. The same effect may hold true for Ada.

Initially, the rationale for validation is the support of the goals of the overall Ada program (which is much larger than just a programming language) and is given additional impetus by the need to provide a mechanism for the protection of the trademark which has been registered for the name. Thus unless it is intended to make the conformance requirements so lax that they can be enforced by "inspection", then a validation mechanism will be required for each element of the Ada program including not only language implementations but also support tools which are inherent to the program.

THE PROGRAMMING ENVIRONMENT MODEL

When considering the validation of a programming environment one must consider the underlying model which is used to construct such entities initially. While it is true that the model which is proposed here is selected on the basis of its adequacy in a

validation environment, this same model can readily be the basis of the development of the programming environments.

The model displayed by Buxton [2] indicates a core-plus-ring structure which logically elucidates, from a functional point of view, the relationships between KAPSE, MAPSE and APSE systems but does not clearly delineate between functional and communications requirements. For example, the diagram of Figure 1 infers the existence of interfaces between elements of the environment(s) but lacks the depth to indicate data-flow requirements which may be superimposed. That is, the model is two dimensional when the problems to be solved are (at least) three dimensional.

An alternative model, though not one which was designed specifically for the design of programming environments is the Open Systems Interconnection Reference Model (OSI) [3] developed by the International Organization for Standardization (ISO) Technical Committee 97, Subcommittee 16. This model is primarily concerned with the modeling of networks and may contain facilities which are too detailed for a direct one-to-one mapping onto a programming environment. However, it is a feature of the model to be able to combine "layers" in order to implement an actual system.

The working environments of Ada developed (and developing) systems may well include OSI-like environments when one considers the complexity that can be created by tasking, multi-processor and multi-targetable systems. Thus the use of this network model will provide a second basis for further considerations of extended environments other than those of "simple" program

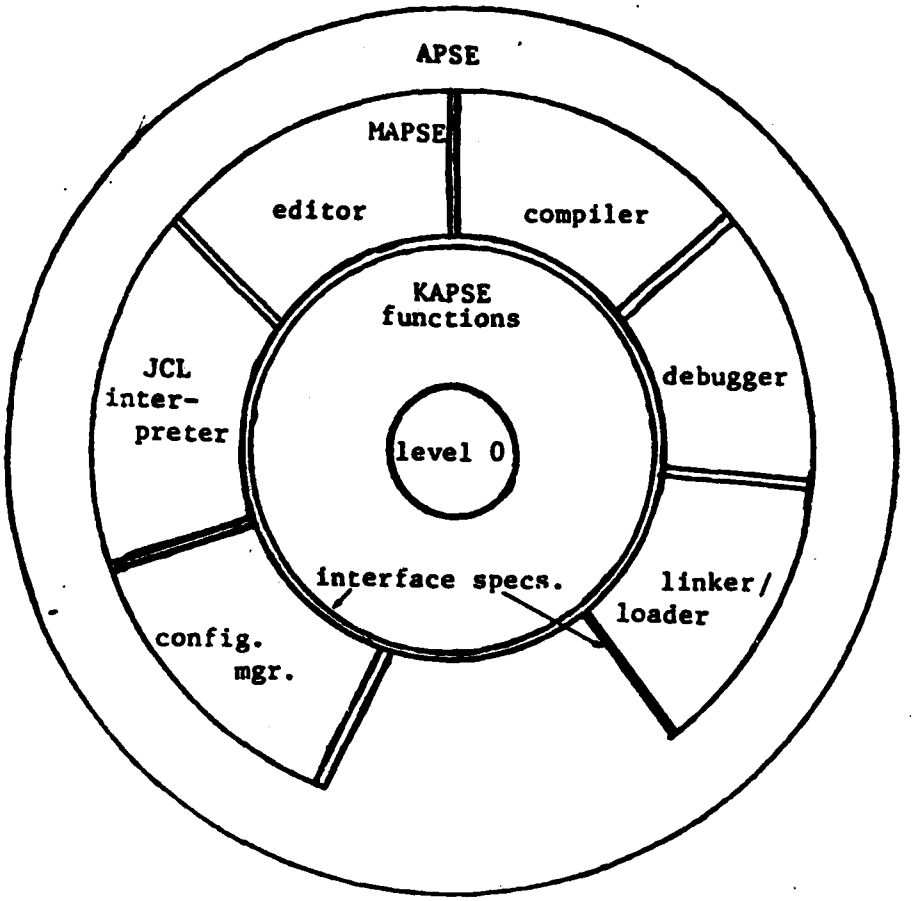


Figure 1. The STONEMAN MODEL

development in a single family of architectures. A salient feature of Ada program/system development must be the portability of software [4] and a natural development from that requirement will be the development of such software in a network environment where both the development and target systems are accessible to the on-line developer.

It has already been hinted [10,11] that there may exist some variations in KAPSE configurations due to the differences in hardware architectures which underlie them; the unfortunate side effect of such developments should not be that those same KAPSEs could not exist in a common (Open Systems) network. A start towards this goal would be the use of a common (standardized) development model.

A third advantage of the OSI Reference Model is the world-wide acceptance of this model for Open Systems Interconnection by network developers. The American National Standards Committee X3 -- Information Processing -- has already accepted this model as the basis for all future standards development work and requires that all proposals for both development projects and draft standards clearly identify how the work fits into this model. Several mainframe manufacturers have announced their intention to build systems which conform to this model and some significantly-sized users have expressed the desire to purchase systems which utilize this architecture.

The OSI Reference Model

*In the concept of OSI, a system is a set of one or more computers, associated software, peripherals, terminals, human operators, physical processes, information transfer means, etc., that forms an autonomous whole capable of performing information processing and/or information transfer. An application-process is an element within a system which performs the information

---

* Extracted from DP 7498, ISO.

processing for a particular application.

The Reference Model contains seven layers:

a) the Application Layer (layer 7); This is the layer in which "real work" of the system is accomplished; the remainder of the layers provide the services by which this layer communicates with other application layers in a network. The application layer interfaces with the outside world.

b) the Presentation Layer (layer 6); this layer provides for the representation of information that application-entities either communicate or refer to in their dialog. The Presentation Layer is concerned only with the syntactic view of the presentation image and transferred data but not with its semantics, i.e. its meaning to the Application Layer.

c) the Session Layer (layer 5); the management of (for example) a terminal session is the responsibility of the session layer. Such responsibilities include the necessary resource management for the support of the application.

d) the Transport Layer (layer 4); the transfer of data between session layers is the task of this layer. For example, if the application needed access to a data-base, then it would be the responsibility of the transport layer to negotiate the data exchange between the session layers which support the specific application and the data base system (which is itself treated as an application-entity in the model).

e) the Network Layer (layer 3); the connection between two nodes of the overall network must be managed by the third layer by providing the services of relaying data between end systems with network connections.

f) the Data-Link Layer (layer 2); the essential element of any network is the data link between them in order to exchange data-link-service-units which implement the network activities.

g) the Physical Layer (layer 1); the physical layer provides mechanical, electrical, functional and procedural means to activate, maintain and deactivate physical connections between systems through the use of the physical media on which it is built.

These layers are illustrated in Figure 2. The highest layer is the Application Layer and it consists of the application-entities

that cooperate in the OSI environment. The lower layers provide

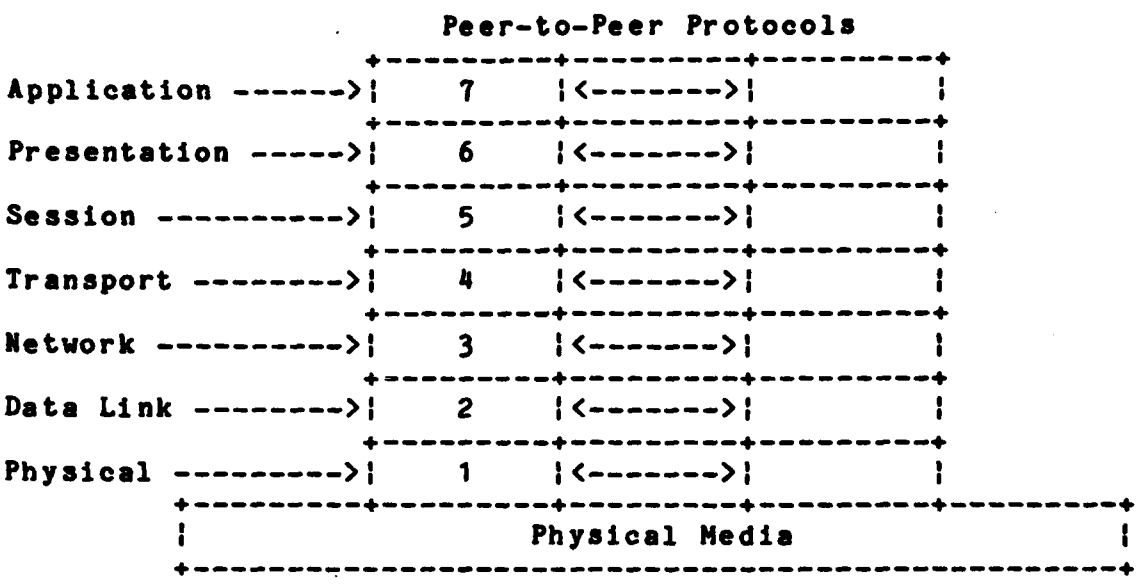


Figure 2. Seven Layer Reference Model

the services through which the application-entities cooperate. An essential element of the OSI model is the clear definition of protocols and interfaces between layers of the model. The primary communication processes in the model are those that communicate with processes which exist in the same level layer as they themselves reside. This is peer-to-peer communication. That is, appplication-layer processes communicate with other application-layer processes in layer 7, networks communicate (in layer 4) with other networks and so forth. This manner of communication is a logical transmission which requires the use of peer-to-peer protocols to implement the data exchange. The logical communication process is physically accomplished through

the (vertically stacked) interfaces between consecutive layers, and each layer provides services to the layer above it while receiving service from its lower layer.

The Application of the OSI Reference Model to an APSE

The original intent of the OSI Reference Model was not to actually represent an implementation strategy but instead to model those elements of a communications environment which need attention in the domain of networks and distributed systems. However such a model is clearly applicable to communications within both so-called "federated" systems and stand-alone environments. Moreover the facilities provided in the layers of the model have affinities with the tasks to be undertaken by other administrative aspects of networking and distributed processors. Thus rather than proposing to add other layers of complexity to the OSI Reference Model it is proposed here that additional features be added to the layers in order to represent the non-communication facilities of these environments.

The OSI Reference Model can be implemented in a variety of manners, one of which would be to combine several layers into one. Thus the model can be applied to the design and implementation of programming support systems. Further it is not required that a layer be composed of only a single process (or processor); in fact it is clear that in many situations a layer will have to be composed of a collection of facilities which provide the support to the layer above, or receive data from lower layers. In an Ada Programming Support Environment, the majority of tools will exist as application-layer processes and

the facilities of the KAPSE will be contained in the session layer. The presentation layer will contain the mechanisms which provide the communication between the application (an Ada program or a tool) and the KAPSE facilities. This can be viewed as an argument to the parameter mapping function and is specified through the specification parts of the packages which constitute the KAPSE facilities. This mapping of the APSE models onto the

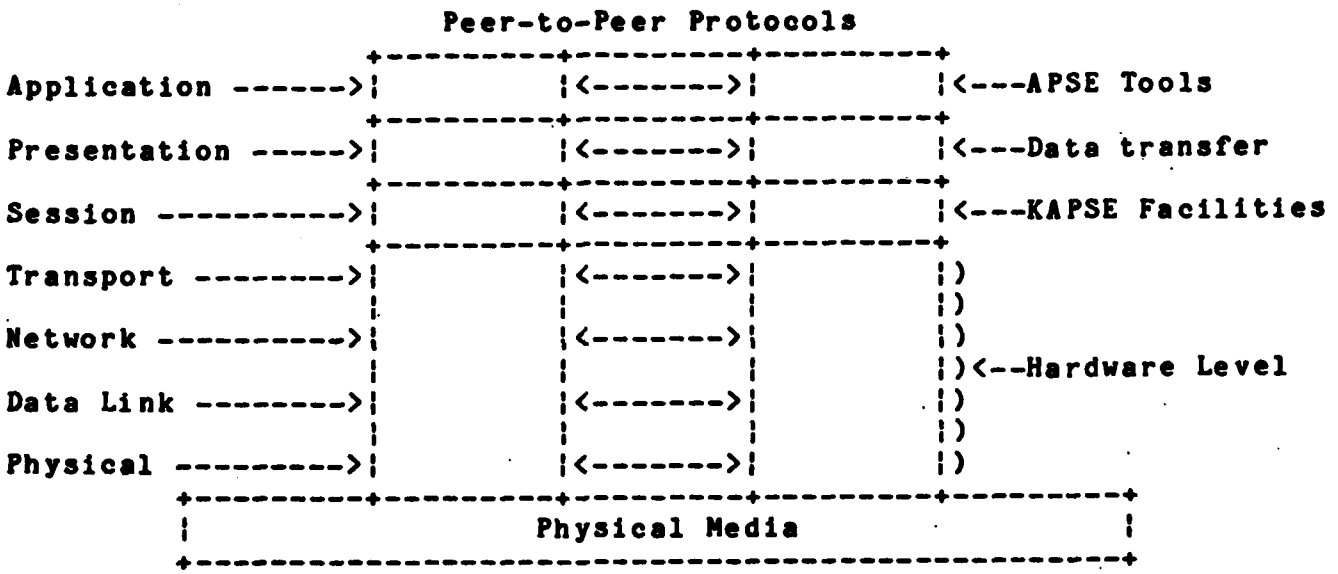


Figure 3. APSE Reference Model

OSI model is shown in Figure 3.

In a "stand-alone" environment, layers 1-4 may be decomposed into a single module which represents the hardware system on which the Ada system is implemented. One other visualization would provide a subdivision of each layer either by the vertical stacking of sublayers or the division of each layer into a set of

elements each of which interfaces with the lower layer. In this latter case, the tools which constitute the application layer are built individually on the presentation layer and have no other means of communication with each other than through the KAPSE (session) layer. On the other hand, it is more than likely that such systems will be required to communicate in order to provide cross-compilation, on-line debugging and similar inter-system activities. Thus, it is imperative that a networking model, such as the OSI Reference Model, be an integral part of the APSE implementation system. This model also clearly indicates where validation systems need to be installed.

VALIDATION METHODS AND PROBLEMS

The problems of validating a Programming Support Environment are akin to those of validating an operating system, a task which itself has not been attempted previously. Not the least of the problems to be resolved is the validation of the collection of tools and facilities which make up the environment but also the communications between those elements when they are integrated into the support system. Let us consider these elements separately.

Validation Elements

The OSI Reference Model presented in the previous section contains two basic elements. The first element is a set of objects each of which performs a specific, well-defined function. The internal mechanism which implements this function is

concealed from view and the object may be readily interchanged with any other object, differently implemented, which performs the same function.

In the ADA context, we may conveniently identify two types of objects: facilities, either KAPSE facilities or other predefined library facilities (e.g., INPUT_OUTPUT), and tools. This distinction between facilities and tools is useful because they are designed with different goals in mind. Facilities are intended to provide generic capabilities out of which a large number of tools could be constructed. Conversely, the facilities are neither concerned with any specific tool nor with the needs or characteristics of any eventual end-user, human or otherwise. By contrast, a tool is intended to provide a specific type of service to an end-user. The nature of this service is derived from the requirements of that user and from the operation of other tools with which the service may interact. The distinction between facilities and tools is also useful from a validation point of view because, as will be seen later, the techniques used to validate facilities are different from those used to validate tools.

The second basic element in the OSI model is the mechanism by which objects interact. An interface is the means of immediate interaction between a tool and a facility or between two different facilities. Recall, also, that the model requires that two objects related via an interface be located on strictly adjacent levels. As an example, an interface may exist between

an editor tool and a KAPSE facility. Also an interface may exist between a library facility and a KAPSE facility.

An interaction between two objects on the same level is a ıieved via a protocol. A protocol is the means by which an object assigns meaning to tokens which it sends to, or receives from, another object. An interaction between two tools, then, involves two protocols - one for each tool. Proper interaction between these two tools is only achieved, of course, when both tools use the same protocol. For example, an editor tool and a compiler tool typically interact through files which are passed between them. The editor creates a source-text file for subsequent compilation. The compiler produces an annotated source-listing, with possible diagnostic information, which may be viewed by a user through the editor. The two protocols involved in this example are manifested in the assumptions which each tool makes about the structure of these files. The following example illustrates how these two tools may fail to communicate properly because they are using different protocols. Suppose the editor tool assumes that the first line of each file which it creates or edits contains the settings for various editor options (line length, search modes, user-defined keys, terminal characteristics, etc.). On the other hand, suppose that the compiler assumes that all lines in a file contain compileable source-text. In this situation, an otherwise valid source-program file produced by the editor would, when presented to the compiler, result in error messages because the first line in the file would not be valid source-text. Furthermore, these error

messages could not be related to the user view of the source text because the first line as seen by the compiler is not visible to the user. In addition, the source-listing file created by the compiler might not even be readable by the editor because the editor expects that the first line of text contains the option settings which will not be the case for files produced by the compiler. More subtle, but equally important, forms of protocols will be described later.

The distinction made between objects and the communication paths between objects provides natural points at which the validation effort may be directed. A complete validation may be structured into two phases. The first phase is the validation of the objects by themselves. Attention in this phase is limited to a validation of the functionality of each object, where the object is treated as an isolated entity. The second phase validates the interfaces and protocols through which the objects interact with each other. In this second phase, attention is directed away from the functionality of the object being validated and becomes focussed on the assumptions which the object makes in its interactions with other objects.

This division of the validation process into two disjoint phases appears to be both natural and technically feasible. There are, however, certain questions which must be further studied. For example: Is this division equally applicable to both APSE tools and KAPSE facilities? Is it always useful to separate the validation into these two phases? Is there a

difference in the order in which the two phases are performed? These questions are meant to suggest the further inquiry that must be made into the question of validating tools and facilities.

Validation of Facilities and Tools

As indicated above, the first phase of the validation effort establishes to what degree an object, considered as an isolated entity, provides the function(s) expected of that object. This first phase of the validation process requires both a proper specification of the object's function(s) and a conformance mechanism. The specification, of course, must be a clear, complete, precise and unambiguous statement of the object's desired function against which the actual behavior of the object may be evaluated. The conformance mechanism is a, hopefully automated, process through which it is ascertained whether the object, in fact, displays all of the functions described in the object's specification. The conformance mechanism is typically a functional testing tool which is driven by a suite of test cases. These test cases, in turn, are derived from an analysis of the specification. There is a large body of literature dealing with the techniques of functional testing [5] and test-case selection [6]. It is critical, however, to realize that these techniques cannot be applied rigorously and effectively in the absence of a properly stated specification.

It is more difficult to write a proper specification for Ada tools than it is for facilities. The difference lies in the fact

that facilities are always expressed as packages and typically contain only a few functions with narrowly defined semantics. For example, common facilities might be "read a single character from a stream" or "create a new process". The use of packages themselves gives a concrete form to the specification of all syntax information for the facility. In a later section, the methods for specifying the semantics of KAPSE facilities will be considered. A tool, however, will typically exist as an Ada program (not a package) which is invoked through the command language mechanism. The tool will typically provide a broad range of individual, but interrelated, functions which may involve extensive user-interaction (e.g. an editor or debugger tool) or may have input with complex syntax (e.g. a compiler tool or text-formatting tool). Unlike facilities whose syntax of use is captured by the package specification, the syntax of tool use is not embodied in any Ada language construct.

The conformance mechanism is also more difficult to construct for tools than for facilities. Given a validation suite for a facility, a specialized conformance-tool can be built which exercises the facility as dictated by the validation suite and which compares the behavior required by the validation suite with the result produced by the facility. Discrepancies between the desired and actual function are detected and reported. The conformance mechanism for a tool is more complicated when the tool involves significant user-interaction. As an example, consider a full-screen editor tool which uses the cursor position to determine a reference location for editing commands.

Presumably the editor tool uses standard KAPSE facilities for terminal I/O. It may also be the case that the terminal facilities are distinct from the facilities for file I/O or interprocess communication. A part of the validation suite for the editor tool would involve perhaps many test cases which exercise the various editing-functions at different cursor-positions (e.g., a deletion/insertion at the beginning/end of the screen/line). One conformance-mechanism is, of course, to use a human operator to perform these test cases. This approach is both expensive and error-prone. To create an automated process which could perform these tests would require not only a conformance tool to drive the delivery of the test cases and check the results of each test but, more significantly, would require some means of intercepting or simulating the terminal I/O. This may either be done by a separate computer system which is attached to the system under test through a terminal port or by the creation of a new terminal I/O-facility, which would allow the output produced by the editor and destined for the terminal to be diverted to the conformance tool itself. Similarly, the input expected by the editor tools would be taken from the output stream of the conformance tool and presented to the editor tool as if it were a terminal stream. Whether this new facility could be easily constructed from the existing KAPSE facilities depends entirely on the way in which those facilities have been implemented and, possibly, on the underlying operating system.

Validation of Interfaces and Protocols

The second basic element, in the Ada environment model, is the means by which objects interact through interfaces, providing communication between objects at adjacent levels, or protocols, providing communication between objects at the same level. The validation of the interfaces or protocols relating a given pair of objects assumes that a validation suite is available which exercises the full range of possible communications. In the following paragraphs the validation requirements for interfaces and protocols are discussed in more detail and their relationship to the validation suite is described.

The critical interface to be validated in the Ada environment is the interface between a tool and one or more facilities. While there may be additional interfaces present within the tool itself, these interfaces are concealed from the view of the tool validator, which is the validation viewpoint taken here. The relevant question then is: "For a given tool how can it be determined whether that tool uses only standard KAPSE facilities and uses them only in a manner consistent with the definition of these facilities?". Dealing with this question is important for two reasons. First, focussing the validation on the tool-facility interface assures that the validation suite is sufficiently robust to exercise the full range of their interaction. Without this focus it would not be difficult to construct an otherwise impressive validation suite, which did not exercise some exceptional circumstances of the tool-facility interface (e.g. a memory fault). Second, even a thoroughly tested tool may have been developed and tested on a non-standard

KAPSE or on a system which incorrectly simulated the KAPSE facilities. In this case the interface validation would reveal the level of conformance with standard KAPSE facilities.

The obvious process for validating the tool-facility interface is by exercising the tool portability. That is, the tool and its validation suite are removed from the development environment and transported to a validated, standard KAPSE. The tool validation-suite is then applied to the tool. If the tool operates successfully in this standard environment then the interface is considered to be validated.

Two implications follow from this validation approach. First, it must be possible to know how extensively and in what ways the validation suite exercised the interface. This knowledge can only be obtained by instrumenting the KAPSE facilities themselves. This seems to imply that a specialized "validation KAPSE" should be developed which contains the required instrumentation. This answer, however, only reveals a further question: "What interpretation can be given to the measurements obtained by the instrumentation?" or conversely "What events should the instrumentation be recording?" The answer to this question involves the way in which the tool interface specifications are themselves stated. That is, the specification of the tool must contain an explicit and complete description of exactly how the tool will use the KAPSE facilities. This specification should include at least the following items:

(1) names of all KAPSE facilities used

(2) name of each procedure, type and data item used in each facility
(3) ranges of parameter values for all IN, IN-OUT, and OUT parameters for each procedure
(4) important sequences of facility usage
(5) error exceptions anticipated

This list is only suggestive of the further thought that must be given to this issue. Only an agreed-upon specification-form can lead to the development of an instrumented validation-KAPSE, which, in turn, is necessary to perform the validation act itself.

The second implication which follows from the approach of interface validation by portability concerns the way in which the validation suite is applied to the tool being validated. Presumably, an automated driver was created in the development environment which performed this task. The role of this automated driver (conformance mechanism) was addressed above. However, to efficiently perform the validation, this driver itself must be transported to the validation KAPSE. Thus, not only must tools be portable, but their test drivers must be portable as well. It may well be made a concrete validation requirement that such a portable, automated driver must be supplied along with the tool to be validated and the validation suite. One difficult problem which arises, and perhaps reflects on how the KAPSE itself should be defined, is that, as noted above, the test driver may need to use modified KAPSE facilities to simulate, or intercept, the tool's terminal I/O without modifying the tool itself. It is also unrealistic to achieve this goal by modifying the validation KAPSE. Perhaps, the

original KAPSE design should provide a mechanism which allows a driver to capture the terminal I/O stream generated by a tool. In any event, some means must be found for resolving this problem.

The validation of protocols between Ada tools is also an area where additional work will be required. Many basic tool complexes possess protocols in one form or another. Typical tool pairs which interact through protocols are: compiler-linker, linker-debugger, librarian-configuration manager, text formatter-device manager, etc. The ways in which tools can be related through protocols are varied. Some of these ways are:

(1) intermediate files
(2) message stream communication
(3) timing synchronization
(4) resource contention

While there may be other ways in which the tool protocol is manifested, the important point is that the tool validation must take into consideration the interaction of the tool being validated with other tools. This wider viewpoint, which encompasses the validation of the protocol, is necessary because the tool is not an isolated entity which functions in a vacuum. Rather tools cooperating through protocols form an integrated entity whose combined function must also fall within the purview of the validation process.

As with interfaces, the validation of protocols between Ada tools is based on an accurate specification of the protocol itself. Two classes of protocols may be distinguished in terms

of their difficulty of specification. Protocols in the first class are characterized by the use of a file to pass a complete body of information from one tool to the next (e.g., the file passed from the editor to the compiler). The communication established by this protocol is uni-directional and the tool which produces the file (the editor) completes before the tool which consumes the file (the compiler) is initiated. The specification of these protocols is no more difficult than detailing a file format and indicating the meaning of each part of the format. This can easily be done using some variation of a data structure diagram [7]. The validation of this protocol is also straightforward.

The second class of protocols are those in which there is a bi-directional communication between two simultaneously active tools (e.g., a debugger tool and an application tool). This protocol may achieve either the exchange of information bearing messages or the exchange of synchronization signals. This second class of protocols is more difficult to specify and to validate. There are at least four possible methods for the specification of these types of protocols. First, there are informal descriptions of the protocol with the usual ambiguity and imprecision which that entails. Second, formal representations, such as finite state machines or petri nets, can be used. In these representations each tool is considered to be in one of several states. The transitions between states of a tool are triggered by the transmission or the receipt of a message or signal. Using these methods it is possible to analytically determine important

properties of the protocol (e.g. deadlock). This method of specification seems to provide a more suitable basis from which the validation suite can be derived. A third possible method for specifying the protocol is by appeal to a predefined inter-tool protocol. For example, a standard debugger-application tool protocol may be defined in advance of the construction of either tool. This predefined protocol may be established as a package whose procedures implement the protocol. The validation amounts to certifying the correct use of these procedures in each tool. Fourth, a protocol "specification" language may be used which contains high level constructs for defining synchronization relationships. This language may be procedural (using, for example, monitors or a CSP-style type of primitive operation) or non-procedural (using techniques like path expressions to describe legitimate sequences of concurrent procedure invocations).

Regardless of the specification technique employed, the conformance mechanism must ultimately apply the validation suite derived from the specification to the tool(s) being validated. Given a tool, A, whose protocol with another tool, B, is to be validated, the conformance mechanism would replace tool B by a previously validated equivalent of tool B or by a stub tool designed to respond appropriately to the circumstances created by the validation suite. Tool A is then exercised as determined by the validation suite and its behavior is compared to that defined as correct by the validation suite.

Since there is a clear distinction between protocols and interfaces, the validation of protocols can be done in the development environment itself. That is, the protocols do not depend on their implementation being achieved by the use of standard KAPSE functions. If, in fact, the protocols are implemented using non-standard facilities that fact will be revealed when the interface validation is attempted. Recall that the interface validation requires that the tool being validated must be moved outside of the development environment. This separation of validation between interface validation and protocol validation is one of the important benefits which is derived from the use of the reference model presented earlier.

Specification of KAPSE Facilities

It was noted above that the validation process is based on a precise and complete specification of the object being validated. In this section possible methods for specifying KAPSE facilites will be considered. One promising specification method, using an abstract-machine, will be illustrated by example.

The specification for each KAPSE facility must include the following details:

(1) syntax
(2) semantics
(3) limits
(4) hidden protocols

Accurate specifications of this information is necessary in order to design a robust validation suite for the facility. The

meaning and role of each of these items in the validation process is explained below.

The syntax information is easy to specify since each KAPSE facility is defined as an Ada package. The package specification itself, containing operation names, type names, etc., serves to define the syntax of the KAPSE facility.

The semantic information is both the most difficult and the most important part of the specification because it defines the intended "meaning" or "function" of the KAPSE facility. The semantic information can be provided in four different ways. First, a natural-language specification of the semantics merely describes in English prose the intended operation of the facility. While natural-language specifications are common (see for example the ALS KAPSE package shown in the example in the next section), it is not possible to verify if the natural-language specification is complete, consistent or unambiguous. Second, there are several formal methods for specifying semantic properties including axiomatic specifications, denotation semantics, or validation assertions. These methods employ a mathematical formalism to define the semantic information, which are precise and subject to mathematical analysis but they are also expensive to construct and difficult for typical programmers and system designers to understand. However, since the KAPSE facilities will be defined only once and implemented numerous times the precision of a formal specification may be worth its cost. The value of a formal specification can also be enhanced

by including an informal, natural-language commentary which increases the understandability of the formal specifications. It should be remembered, however, that the specification is ultimately the formal description and not the additional commentary since the validation suite would be constructed in agreement with the formal specifications. A third method for specifying the KAPSE semantics is through an abstract-machine. This abstract-machine contains as primitive objects the elements appearing in the KAPSE environment (e.g., streams, flags, histories, processes, etc.). The specification of a KAPSE facility is given by writing a "program" for this abstract-machine which, if executed, would perform the function intended for the KAPSE facility. That is, the semantics of the program are the semantics of the KAPSE facility. An example of this approach is given in the next section. By comparison with the other methods, the abstract-machine approach is more formal and concise than the natural-language specification, easier to comprehend than the formal methods, but is less precise than the formal methods because the semantics of the language used to write the "program" must now be defined. This later problem can be minimized by using Ada itself as the language in which to write the program for the abstract-machine. The fourth, and final, method of semantic specification is by example. In this method the validation suite is constructed first and, by definition, anything which acts in accordance with that validation suite implements the intended semantics. In this method the semantic information is described implicitly.

The importance of the KAPSE semantics to the validation effort implies that several of the four specification methods described above may be used in conjunction to specify the KAPSE semantics. By using several specification methods concurrently it is possible to satisfy the conflicting demands of precision and understandability of the specification. For example, a complete specification might contain either an abstract-machine program or a formal mathematical description supplemented by a natural-language commentary and illustrated by a validation suite. Disagreements among these three descriptions can be avoided by arranging each part of the commentary as an expansion of a single, specific part of the formal description rather than creating the natural-language description separate from and differently organized than the formal specification. Similarly, each case in the validation suite should be tied to a specific part of the formal specification and commentary. In this way, the different levels of description are reinforcing and provide both high precision and ease of understanding.

The third part of the KAPSE specification must include details of any limits which are applicable to the facility being specified. Such limit information describes the sizes of objects (e.g. identifier strings, maximum file size, maximum number of entries, and maximum number of processes) and the number of times that operations may be repeated. The importance of this limit information to the validation process is illustrated by the following example. Suppose that an APSE tool is designed to create 10 subprocesses and to produce a file that is 10 megabytes

in length. Without careful attention being paid to the limit specifications it is easily possible to implement this tool on a KAPSE supported by a large machine which uses only the functions provided by the KAPSE on that machine. However, when that tool is transported to a KAPSE supported by a much smaller machine, the tool may be unable to operate because of the more limited environnment of the second system. It is important to see in this example that the impediment to portability is not a function of the KAPSE facilities, but of the limits which circumscribe the extent or repetition of that facility. Since the KAPSE may be implemented on machines with widely-varying machine resources, it would not be unusual to consider validating a tool relative to a defined set of limits. While this does not make a tool any more or less portable, it does bring into clear view the degree of portability of that tool. Such a conscious statement of the limit specifications may also serve to control the tool design if a highly-portable tool is the design goal.

The fourth, and last, part the KAPSE specification is a description of the "hidden protocols". In a previous section the issue of validating the protocols between APSE tools was discussed. On the surface there does not appear to be a similar validation requirement for KAPSE facilities because each KAPSE facility appears an an independent function. However, beneath the level of the KAPSE interface many of the KAPSE facilities are, in fact, related through data objects in the implementation of the facilities. These underlying connections, or protocols, are "hidden" from the view of the validator because they are

implementation dependent and cannot be directly observed - they can only be validated indirectly by observing the joint behavior of those KAPSE facilities interrelated by a given "hidden protocol".

For example, the procedure ANNOTATE in the ALS KAPSE contains the following specification:

> "This procedure adds annotation to the derivation record. The text parameter contains the text to be added. ... When a file is closed and receives derivation information, the annotation is placed in the file's deriv_text attribute."

In this case the annotation which is supplied as a parameter to the ANNOTATE procedure is retained in some implementation object until the file being annotated is closed. At that time the text is retrieved from the implementation object and copied to the file's "deriv_text attribute". Notice that the existence of this implementation object is only implied by the specification. This implementation object embodies a "hidden protocol" since it allows a lateral flow of information between objects at the same level -- in this case between KAPSE facilities.

Since the validation process must assess the behavior of facilities related by such "hidden protocols", it is important that the KAPSE specifications make visible the relationships between KAPSE facilities. This visibility can be achieved by a simple mechanism. If there are N KAPSE facilities, an [NxN] table can be constructed where each entry in the table describes the protocol connecting a pair of KAPSE facilities. For example,

using the ALS procedure described above, the corresponding entry in the hidden protocol table, named HPT, might be:

```
HPT[ANNOTATE,CLOSE]  =

        "annotation information supplied to the ANNOTATE
         procedure is added by CLOSE to the deriv_text
         attribute."
```

The validation suite would then contain a set of test cases which evaluates the correctness of the interactions identified by the KAPSE hidden-protocol table.

Example of a KAPSE Specification

The following example is taken from the preliminary design specification of the ALS KAPSE. The comments which follow are intended to illustrate a method of specification, using a program for an abstract-machine, which can be more precise and more comprehensible than a natural-language specification. This example, is not meant to be critical of the design or specification developed by SofTech [8] but reveals, we believe, common and fundamental problems in KAPSE specifications which all implementors face. The brief example concerns two procedures defined in the FILE_DERIV package: the procedure ANNOTATE and the procedure CITE_INPUT. The natural-language specifications for these two procedures are as follows:

ANNOTATE:

```
    procedure annotate

        (  annot_text : in string_util.var_string_rec;
                result : out io_defs.io_result_enu;
        result_string : in out string_util.var_string_rec
```

```
    );
```

This procedure adds annotation to the derivation record. The text parameter contains the text to be added. A null value for the text parameter is a special case, causing all current annotation to be cleared. When a file is closed and receives derivation information, the annotation is placed in the file's deriv_text attribute. The result parameter indicates the success or failure of the request.

CITE_INPUT:

```
procedure CITE_INPUT

    ( stream        : in io_defs.stream_id_prv;
      cite_flag     : in boolean;
      result        : out io_defs.io_results_enu;
      result_string : in out string_util.var_string_rec
    );
```

This procedure sets or clears a flag indicating if an input file is to be cited in derivation histories. The stream parameter indicates the stream assocated with the input file. The stream must be associated with an environment database file opened for ''in'' or ''inout'' access. The cite_flag parameter indicates if the flag is to be set or cleared. If it is true, the input file will be cited in subsequent derivations, if it is false the input file will not be cited. The result parameter indicates the success or failure of the request. The flag for any given input file may be set and reset as often as necessary. the value of the flag at the time a derivation is being written determines if the file is cited in that particular derivation or not. (Derivations are written only when output files are closed.) To be cited in a derivation means that the file will be referenced in the output file's derived_from association. Input files that are not cited are referenced by the output file's other_inputs assocation. In other words, if an input file is "cited" in a derivation the derivation information is "hard", i.e., the derivation_count attribute of the input file is incremented when the derivation is written to the output file. Input files that are not "cited" in the derivation are still mentioned in a "soft" manner, i.e., the derivation count of the input file is not incremented.

Notice that in the above specifications, certain basic objects are referred to: derivation_records which may be added to or cleared, input/output streams which are of different types and

have derivation and record flags which may each be set or cleared, and string used for returning result information. Since these objects are fundamental entities in the universe being defined by the KAPSE, these objects are explicitly defined in the program of the abstract-machine. Thus, the following abstract program contains the declaration of three objects: derivation_record, string, and io_stream. The bodies of the two procedures ANNOTATE and CITE_INPUT are also given. Additional comments on this abstract program specification are given after the abstract program.

Abstract-Machine Specifications for the ALS package FILE DERIV:

```
OBJECT derivation_record HAS
    clear, add : OPERATIONS;
    capacity   : ATTRIBUTE;
end derivation_record;


OBJECT string HAS
    "=",":=" : OPERATIONS;
    length   : ATTRIBUTE;
end string;


OBJECT code is ENUMERATED TYPE;



procedure annotate(
    annot_text:  in string;
    result     :  out code;
    result_string: in out string );
begin
    if annot_text = null
    then clear (derivation_record);
    elseif derivation_record'capacity < annot_text'length
    then begin
    result := "failure code";
    result_string := "insufficient space"
    end
```

```
        else begin
        add(annot_text,derivation_record);
        result := "normal code";
        result_string := "success::
        end;
        endif;
end;


OBJECT io_stream HAS
        access_type, file_type : ATTRIBUTES;
        OBJECT derivation_flag HAS
        set, clear : OPERATIONS;
        end derivation_flag;
        OBJECT record_flag HAS
        set, clear : OPERATIONS;
        end record_flag;
end io_stream;
```

```
procedure CITE_INPUT(
    stream        : in io_stream;
    cite_flag     : in boolean;
    result        : out code;
    result_string: in out string);
begin
    if stream'file_type /= "environment database"
    then begin
    result := "failure code";
    result_string := "not environment database file";
    end;
    elseif stream'access_type /= in or
           stream'access_type /= inout
    then begin
    result := "failure code";
    result_string := "incorrect access type";
    end;
    else begin
    result := normal;
    result_string := "success";
    if cite_flag
    then set(stream.derivation_flag);
    else clear(stream.derivation_flag);
    end;
    endif;
end CITE_INPUT;
```

Several points are of interest to note in the above example. First, in the specification of the ANNOTATE procedure, the fact that insufficient space is one of the possible outcomes of a call on this procedure can only be determined in the original specification by examining the possible values of the enumeration type which defines the result parameter. However, in the abstract-machine approach, this fact is clearly visible. Second, by comparing the natural-language specification of the CITE_INPUT procedure with the abstract-machine specification it appears that much of the specification given in the former case is misplaced because it has little to do with the operation of the CITE_INPUT procedure but it has more to do with the close operation and its

effects. Third, the natural-language specification only implies the nature of the basic objects in the environment whereas the declaration in the abstract-machine approach identify the attributes and operations for each such object. For example, to determine the total characteristics of an io_stream, it is necessary to read several parts of the specification of CITE_INPUT and other procedures while there is a single place in the abstract program which provides this information.

The abstract-machine approach is certainly far from complete and there are many unanswered questions regarding its use. However, it is put forward in this report to indicate a possibly fruitful avenue of research which would permit the KAPSE facilities to be specified in a precise and comprehensible manner.

Validations of Run-Time Environments

The purpose of APSE's (and their subsets) is to provide a standardized, programmer-portable, environment in which Ada programs can be developed, compiled, debugged and tested. In the latter situation it will be necessary to construct model run-time environments in which to perform these tasks since it is unlikely that the actual environments will be secure enough to permit on-line testing.

"Stoneman" [2] suggests (section 2.B.11) that "the KAPSE is a virtual support environment for Ada programs", but it our contention that this should be more correctly stated as "the KAPSE is a virtual support environment for the development of Ada

programs". In the model, and as shown in Figure 4, the KAPSE is replaced by a run-time environment which contains the support facilities defined by the Ada Language Reference Manual (LRM) and the necessary PRAGMAS. It is likely that there will be some overlap in the facilities provided in the run-time environments and the KAPSE, but it should not be overlooked that these environments have disjoint members. Thus, there are two further validation activities which must be considered:

(1) the validation of run-time environments as a separate activity from general compiler-validation, and

(2) the validation that the model or test environments not only conform to the general validation requirements (1) above, but that they also correctly "mirror" the run-time environments which they are intended to model.

This latter requirement is necessary since it can easily be anticipated that actual environments will contain facilities which, while they themselves are conforming programs, are peculiar to and modify the general run-time environments required by the Ada LRM. On the other hand, this requirement for validation is not one which should be imposed on a general validation organization, but instead should be part of the requirements contained in contractual agreements with software vendors.

At run-time, the reference model contains not the support environment elements but instead the actual Ada program at the application level, the necessary argument-parameter transfer facilities in the presentation layer, and the run-time support facilities in the session layer as shown in Figure 4.

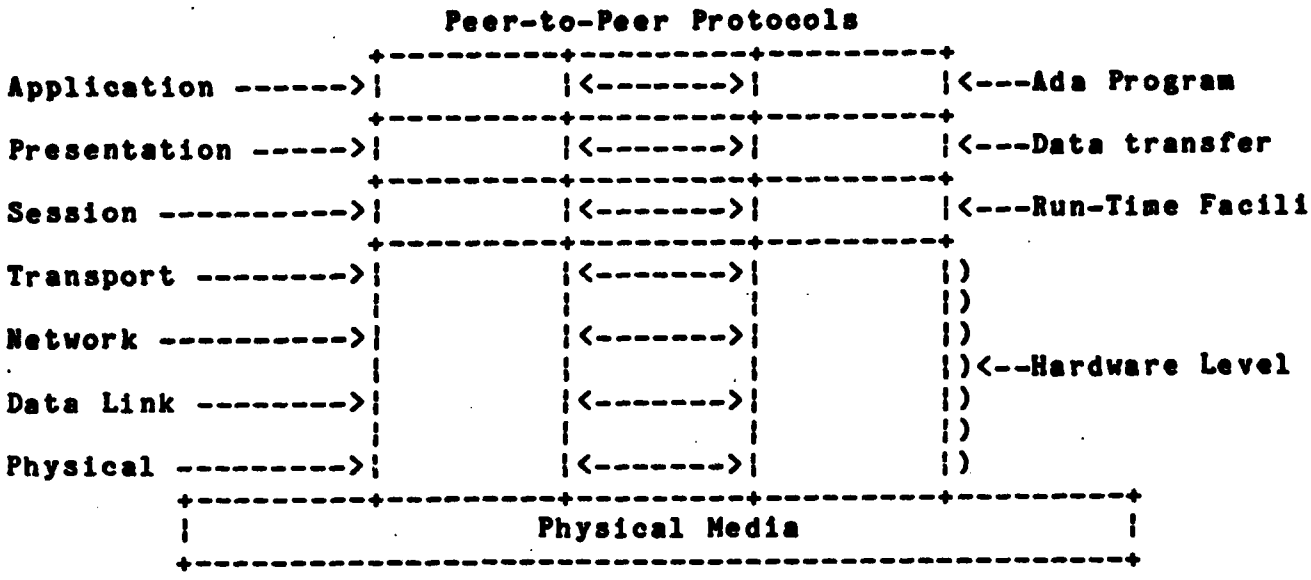


Figure 4. Run-Time Support Model

VALIDATION OF KAPSES AND THEIR FACILITIES

The validation of a programming support environment can primarily be considered in two stages: the validation of the KAPSE and its facilities and subsequently the validation of the tools which are contained in either a MAPSE or a "full" APSE. On the basis of the Reference model, the validation can be considered as the validation of each layer. Within a layer it is necessary to show that the processes contained therein fulfill the service requirements of the higher layer, or in the case of the application layer (7), that the application requirements are met. Between layers it will be necessary to show that the interface requirements are met, and from any layer, communication

with a peer-layer is in conformance with the protocol requirements of the system. This latter situation will occur when (for example) a debugging tool exists at the application level and is monitoring or debugging an application elsewhere.

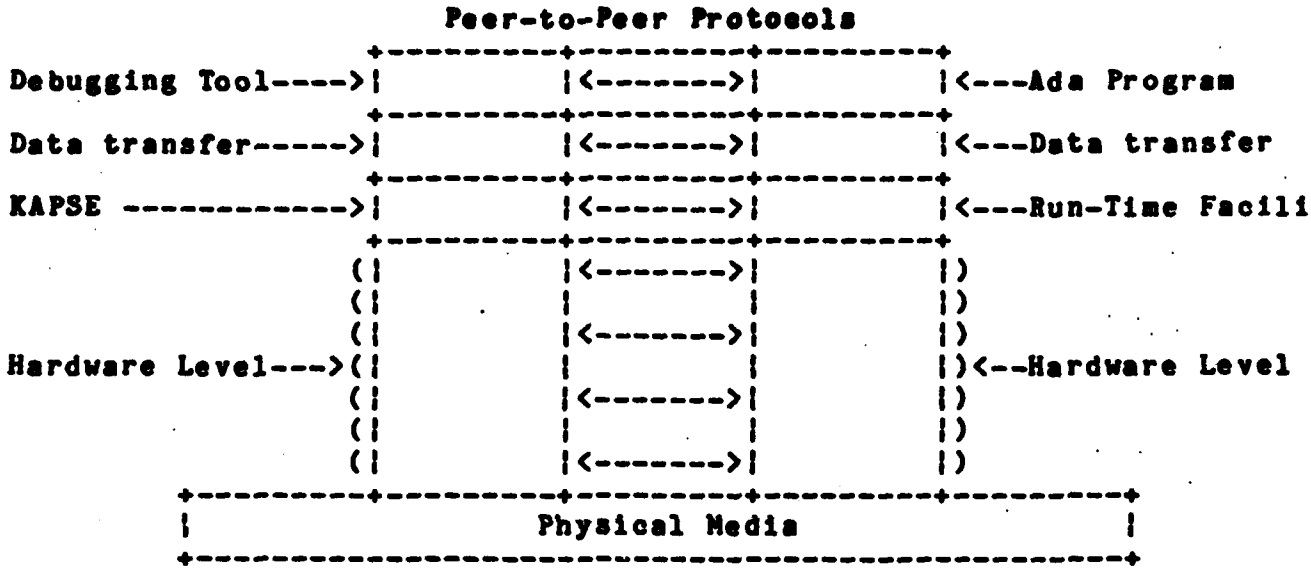


Figure 5. The Debugging Model

This situation is shown in Figure 5. In this model it is assumed that the two systems are separate and that communication takes place through the physical media; in a cross-compilation and debugging environment this model would require extensive facilities to "make sense" of the debugging information which is exchanged between the two environments. Similar models, which all exhibit the same layered characteristics, can be constructed for cross-compilation, data-base access, and multiprocessor systems. Thus it is clearly apparent that a generalized

validation-schema can be developed by the use of this reference model.

KAPSES AS ADA PACKAGES

The STONEMAN requirements [2] specify that:

> "5.E.1 The KAPSE shall implement interface definitions which shall be available to APSE tools. Such interface definitions shall be given in the form of package specifications in the Ada Language.
>
> 5.E.2 Interface definitions provided by the KAPSE shall encompass
>
> (i) the primitive operations that the KAPSE makes available to APSE tools; these include any operations that may be necessary to supplement the facilities of package INPUT_OUPUT ... in order to allow an APSE tool access to all the functional capabilities of the database,
>
> (ii) the abstract data types (type declarations plus operations) that are required to interface the various stages of compilation; these include the data types that are produced by a compilation stage for later use by analysis, testing, or debugging tools."

This implies that the process of generating a support environment (whether at the level of a MAPSE or an APSE, but definitely above the level of a KAPSE) is to possess a "standard" library of KAPSE facilities which exists as a collection of package specifications and bodies which are used during the compilation of the tools. These tools are then added to the library and new tools, added subsequently, build on this foundation until the "whole" is tied together through some command-language or access-system.

This will require that the KAPSE facilities exist in a form which is accessible to the compilation system (which of course resides at the level of tools) as a library. However, since the

facilities have to interact with the hardware system and may be distinctly implementation dependent, it is possible that the facility bodies will not be implementable in Ada itself! Thus it will not be possible to simply validate the facilities as Ada programs primarily; they will only be subject to functional validation. To accomplish both the accessibility of the facilities to the compilers, the tools and the validating processors, it will be necessary to establish "internal standards" for the representation of the bodies and their interfaces with their own specifications. Interfaces between the specifications and the tools which use the KAPSE facilities must conform with the other standards for the Ada language level of interaction. That is, whatever the compiler generates (or is required to generate, depending on which is implemented first) to facilitate the interaction between USE statements and library packages will also be applied to the non-Ada-generated facility bodies. Such standards could well be defined for each implementation but it would be advantageous to specify such interfaces at some common language-level such as through the intermediate-language Diana, or at the very least, through a PDL implemented as the specification part of KAPSE packages.

One of the drawbacks to the current definition of the Ada language is the inability to place assertive statements which depend on values associated with parameters in package specifications. There is a modicum of this ability in the inclusion of constraints in type declarations and an inkling of this ability appears in discriminants in record specifications.

An extension of this ability to include assertive statements connecting the input and output parameters of procedures and functions would provide an on-line validation facility which would greatly enhance the verification of conformance in KAPSE specifications.

THE USE OF LIBRARIES AND THE USE FEATURE

The concept of implementing a KAPSE as a library of packages gives rise to a facilities validation-procedure which is the initial test-suite in a process which ultimately encompasses the Minimal APSE set (MAPSE). That is, the specifications of the facilities (which actually prescribe the interfaces with the tools) should be defined in a standard KAPSE document in the programming language Ada, and the functionality of the facilities can be specified in a PDL. However, it will not be possible to require that the specifications be validated in a character-by-character comparison since:

(1) there should be no requirement that the authors of a KAPSE maintain the same naming-conventions for declarations except to maintain a homomorphic relationship between entities, and

(2) the authors may need to add private parts to the specification in order to accomodate other unique procedures (functions and packages) which are not a part of the required facilities set.

On the other hand, validation of the interfaces between a tool and the facility should be capable of being conducted as an integral part of the validation of the facility itself, particularly if the facility is designed (specified) to be responsive to changes in individual parameters and these

responses can be readily ascertained. Thus a facility should be specified in conjunction with a test tool which would exercise its capabilities. Such a validation tool should be a part of the KAPSE description, and be a valid Ada program which can be simply compiled and executed as a stand-alone test of the facility. Overall testing and validation which would validate the interactions between facilities and the interdependence of facilities would be separate*.

COMMUNICATABILITY VERSUS PORTABILITY

The validation of tools requires that tools be regarded as separate, individually defined entities, each with its own specifications document. For example, the compiler is a tool which has its own Language Reference Manual and its own validation suite. By comparison therefore, it will be necessary to write specifications for the editor, debugger, etc. and each specification will have to include the necessary conformance statements by which a validation suite can be constructed and be judged relative to those conformance statements.

It would appear, from the Stoneman requirements [2], that it is necessary that tools be written in the Ada language (see sections 2.B.10 re transportability; 2.B.11 which can be interpreted as

---

* If facilities are defined as independent complete entities, then the intricate interdependence of facilities can be minimized. If one the other hand the implementer desires to "extricate" common features of facilities, this interaction would not be apparent to a tool and thus not subject to validation.

saying that it refers to tools including those written in Ada, and 4.E.3 which asks that tools be "written in Ada") but it is not a requirement of current contracts that the fundamental tool, the compiler, is written in Ada. The only requirement should be that the tools have interfaces (with the KAPSE) which are "standardized". Thus tools act and perform in the same manner as Ada programs but are not required to be Ada conforming programs in their source form.

By the model which has been proposed here, tools may communicate with other tools only through the KAPSE; this can be accomplished best through the provision of a "pass-through" facility in the KAPSE. However, indirect communication is possible through the data-base facility (see Stoneman section 2.B.4). This ability to communicate also provides an alternative to portability in an open systems environment. That is, while the "Stoneman" requirements [2] specify the portability of tools (2.B.11), this can only be achieved at the source-code level between differing architectures; however, the ability to communicate with tools which exist at other nodes of a network will effectively resolve this problem for non-Ada tools. This concept of tools which are implementation-restricted, because of their non-use of Ada, can be characterized by distinguishing between Stoneman-conforming tools which are Ada written and transportable, and APSE-conforming tools which are non-portable but accessible through a communications system. That is, portability can be replaced by communicatability.

KAPSE IMPLEMENTATION STRATEGIES -- EFFECT ON VALIDATION

Although programs in the Ada language may be portable, tools that are written using operating system facilities not directly included in the language would be portable only if those facilities are identically implemented on each machine. In an APSE, KAPSE-facility-specifications provide the interface between the underlying operating system and the support-environment tools. To gain the highest degree of portability of APSE tools, of Ada programs, and of APSE databases, the Department of Defense has promulgated as a distinct goal the evolution of a single KAPSE-interface [9]. Despite this goal, two distinct KAPSE interfaces exist as part of APSE development efforts. The KAPSE of the Army-supported Ada Language System (ALS) and of the Air-Force-supported Ada Integrated Environment (AIE) both have elements in common, but are not compatible for the purposes of transportability.

Lyons [10] details seven likely paths of KAPSE evolution, given that different KAPSE-interfaces currently exist and that some degree of transportability will continue to be needed as evidenced by the memorandum of agreement [9]. Two of the most likely alternatives, the standardization of more than one existing KAPSE and the design and standardization of an entirely new KAPSE-interface, are interesting when examined from the point of view of validation for transportability. Both alternatives have drawbacks and advantages over the other, but from the pragmatic level, satisfying the memorandum of agreement will require the design and standardization of a machine-independent

set of KAPSE facilities. These two alternatives are discussed below from the perspective of validating for transportability. A third alternative, a mixed approach, is also discussed as a practical implementation strategy that allows for transportability.

N-Standard KAPSES's

Momentum is already well-established along the path toward N-standard KAPSE-interfaces. The evolution of a small number, presumably one for each Armed Service, of accepted KAPSE's characterizes this path. Each of the KAPSE's is likely to undergo a de facto standardization process as multiple implementations are developed. To assure portability of tools within one of the standard KAPSE's, i.e. across implementations of the same KAPSE, the development of a validation suite for KAPSE services would be required. In light of the two currently-existing and distinct KAPSE's, the next step in this path will be the proliferation of these systems by multiple KAPSE-implementations. This is certainly anticipated for both the ALS and AIE through rehosting of their KAPSE facilities.

Supposing that a standard for each of the KAPSE's did evolve, validating the transportability of tools across implementations could proceed in one of two ways. The first approach parallels the big-bang approach to testing by submitting the entire set of rehosted KAPSE-facilities to an extensive validation-suite. Doing so would ascertain that any tool developed on the rehosted KAPSE could be transported to another APSE, and that any tool

developed on another validated implementation of the KAPSE could be transported to the rehosted APSE. This gives the appearance that one validation effort for all the facilities provided by the KAPSE would assure the transportability of tools, but in reality such would not be the case. In almost every instance of a tool to be transported, there is another form of interface that must be validated in the new environment. Very few tools do not themselves communicate to other tools either directly through KAPSE facilities or through commonly-accessible data bases. For example, consider a text editor that is to be transported from one APSE to another. As mentioned above, intertool dependencies (or protocols) must be validated whenever tools that depend upon or use other tools are transported in isolation. Aside from a direct interface with KAPSE services through procedure calls, the editor has indirect interfaces, or protocols, with other tools accessing a text file created by the editor. It may be the case that the APSE's compiler expects that Ada source-files to be in a special format somewhat different than other text files. Although the big-bang approach allows the functionality of all KAPSE services to be tested at once, there exists a need to validate the interaction that transported tools have with other tools.

The second approach is essentially divide-and-conquer. Each time that a tool is to be transported to a system, the specific facilities used by the tool and the intertool dependencies are validated. In most cases it would not be necessary to validate the entire KAPSE to transport a single tool; instead, validation

of only those facilities used by the tool would be required. Although a particular implementation's KAPSE-facilities may be validated several times, the advantage of divide-and-conquer is that test cases originally used to validate the functionality of the tool augment the validation of the specific KAPSE facilities.

In general, portability of tools across different implementations of one of the N-standard KAPSES poses no more problems to validation than there exists with just one standard KAPSE. Although validating the functionality of the KAPSE facilities is relatively straightforward through validation suites, there is certainly the additional overhead of standardizing N-different KAPSE's and developing and maintaining N validation-suites. The most damaging drawback, however, is that nothing can be done to provide for the portability of tools from one of the standard KAPSE's to another. Returning to the concept of one standard KAPSE for each Armed Service and realizing that several contractors may work for more than one Armed Service, they would be required to maintain and use different, possibly radically-different, Ada-environments. Additionally, many programs are identified with more than one Armed Service and the software produced on these programs must integrate into each involved Armed Service. To elucidate the options available for validating transportability across standard KAPSE's, suppose that a tool or group of tools are to be transported from one APSE, with KAPSE-A, to another with a different KAPSE, say KAPSE-B. Since it is very likely that the underlying KAPSE-facilities being used are radically different,

one must either simulate the original KAPSE or alter the tool to achieve mobility.

Simulation of KAPSE-A facilities using those provided by KAPSE-B would provide a relatively-simple transporting mechanism that would not require tool modification. The subset of KAPSE-A facilities that are used by the tool could be implemented using either the facilities of KAPSE-B or the host-level operating system. Once the KAPSE-A facilities had been simulated they could be validated using the relevant portion of the suite for KAPSE-A. This is, however, a rather naive view of the requirements for moving tools from one standard KAPSE to another. In the most common situation, major sections of the target KAPSE (KAPSE-A in the above example) would have to be simulated for even the simplest of tools. For example, suppose that the tool to be moved performed file manipulation. Because of the strong dependence of one file-control primitive upon others, all file-control services would have to be simulated. This strong dependence filters up to the tool level as well. It is likely that a single isolated tool could not be transported from one APSE to another unless that tool did not interact with any other tool on the system. A more obvious drawback of this approach is that it is generally quite inefficient to simulate one operating system's primitives in terms of those of another. It appears that the technique of simulation of one set of KAPSE facilities in terms of another set is reasonable only for transporting very simple tools without strict performance-constraints, and the technique can take little advantage of validation efforts already

expended on the involved kernels.

The other approach that can be used to move tools from one standard KAPSE to another is to alter the tool itself to accommodate the new set of KAPSE facilities. Unfortunately, the only way that validation can aid in the process is when a set of test cases has been established to validate the functionality of the tool. Such a set of test cases can be used, with minor modifications, to test the new version of the tool. A distinct disadvantage to changing the tool to transport it is that the effort required to effect the move is highly variable and dependent upon the original design and implementation of the tool. Tools whose KAPSE service-calls are few in number and isolated to a specific section of code have an obvious advantage. Even though well-written tools, independent of all else, are easier to transport, leaving such a large portion of the burden for transportability in the hands of the tool implementor is not advisable.

If the current momentum is not altered, more than one de facto standard KAPSE may evolve. In this event, the relationship between the Ada language and its supporting set of operating system facilities would be little changed from the current relationship between high-level languages and already existing operating systems. A major software-engineering advancement of the Ada Program will be significantly diluted, and only pure Ada language programs will be transportable. This problem has been addressed in a KITIA position paper [11] in which the problems arising from the current Department of Defense efforts are

identified and recommendations are made for a logistically-sound path leading toward a resolution. Although they do not analyze how multiple standard KAPSE's affect the cost of transportability, their paper certainly supports our thesis that validating the KAPSE services to assure transportability can be practically applied only to moving tools between different implementations of the same KAPSE.

A Single Standard KAPSE

KAPSE validation for transportability between different implementations of the same KAPSE is no different for the scenario of N-standard KAPSES than it is for a single KAPSE. Approaches to this validation are now discussed in the context of a single standardized set of KAPSE facilities. Several alternative paths may eventually lead to a single KAPSE-interface. One of the two existing KAPSE's could be selected to be the standard, or parts from each could be selected. From the point of view of pure transportability, the desired route would be to design an entirely-new set of KAPSE facilities based on the Stoneman requirement [2], experience from the ALS and AIE implementations, and forseeable tool-requirements. Such a design could be performed with ease of validation, machine independence, and rehostability on various architectures as primary design-criteria.

As mentioned above, two different strategies could be used to validate that tools written using KAPSE facilities could be transported. Using the big-bang approach, a suite to test the

functionality of all routines making up the KAPSE interface can be developed and used to test new implementations for compliance to the standard. This form of test is presumably large and quite time-consuming to use. The advantage, however, is that early in the re-implementation of a KAPSE several of the subtle errors are encountered and corrected alleviating possible data or program errors that might otherwise be promulgated through the environment's tools. The test suite itself could take the form of an Ada tool using KAPSE services. Thus, KAPSE facilities defined through Ada package specifications along with their machine-dependent implementation in the form of package bodies could be tested by executing the validation tools. The other approach, divide-and-conquer, validates all that is necessary when transporting. Each time that a tool or group of related tools are to be transported, four different levels of validation must take place. The functionality of the rehosted KAPSE-facilities must be validated, each tool must be checked to assure that it uses only standard KAPSE facilities, and shared protocols with different tools must be validated.

(1) Validating the necessary KAPSE-facilities. All KAPSE facilities used by the tool to be transported must be considered. If we assume that a validation suite exists to test the functionality and interactions among KAPSE facilities, that suite would be employed to validate the facilities used by a tool in the new environment. Certainly it is not necessary to revalidate facilities that have already been shown to conform. But, it is necessary to establish that the facilities used by the tool in

its home environment have the same meaning as those in the new environment. To do so it may also be necessary to establish the conformance of the facilities in the home environment of the tool. Additionally, if a tool uses only a small number of a set of highly-related facilities then it may be necessary to validate the entire set as they exist in the new environment. An example of this is the set of KAPSE facilities that manipulates the intermediate form of an Ada compilation-unit. Included in the set are routines to build the intermediate form as well as those needed to access an already-existing intermediate structure. If a tool that is to be transported uses only those facilities for building the form it may be necessary to validate the conformance of the entire set of facilities.

(2) Tool use of KAPSE-facilities. It may also be necessary to assure that a tool uses only standard KAPSE facilities. Non-standard facilities may be hidden within the tool in various forms. One such form might be that a certain KAPSE routine has an additional parameter that is implementation-specific. Although non-standard usage such as additional parameters or new routine-names may be easily detected with static checks, a more difficult deviation would be the use of a non-standard argument to a standard parameter. The use of enumerated types for values of parameters to standard KAPSE facilities would ease this problem, but since some procedures have parameter values for which type-enumeration would not be possible, the problem would still exist.

(3) Tool to tool interactions (protocols). APSE tools that do

not communicate with other tools either directly or through common data-bases are rare. We call such communication an intertool-protocol, and consider it an important part of tool transportability. If a tool that communicates with other tools is moved in isolation, i.e., without moving the related tools, then the tool validation must include tests examining the tools communication in its new environment. Tools such as compilers, linkers, and debuggers are so highly interrelated that transportation of only a single tool of the group would not be practical. This is one instance where the protocol between the tools is so complex that the effort required to transport a single tool would exceed the advantage gained. Tools such as a general text-editor, however, have a protocol with other tools simple enough to allow for isolated transportation of the tool.

In terms of the Open System's model of an APSE presented above, the validation of the KAPSE facilities and the tools use of those facilities is nothing more than a validation that all levels below the application are the same, at least as seen by the tool, in the old and new environments. Validation of a protocol as described above, checks at the same level as the tool to assure that interactions at that level are the same in the new environment as in the old. From the standpoint of the cost required to validate for transportability, the alternative KAPSE evolution leading to one standard KAPSE should be independent of the means used to arrive at a single KAPSE. If the criteria include, however, hostability on various architectures, the most appropriate path to a standard KAPSE is a new design based on

already-existing systems and the needs of validation for transportability.

A Mixed Implementation Strategy

A final implementation-strategy deviates slightly from those above by allowing the KAPSE supporting an individual APSE to contain both standard and nonstandard features. Although the mixed approach is not optimal from the standpoint of transportability, it does provide an alternative in which transportability can be achieved, and permits flexibility within each installation. The approach centers around a minimal set of functionally-complete KAPSE-services which are machine-independent. As a standardized set of facilities they would be implemented and validated on each installation of an APSE, and tools would be written based upon these facilities. The term "minimal" is used to describe the KAPSE to indicate that a truly machine-independent set of facilities may not be efficiently implementable on all systems upon which the KAPSE is to be hosted. Consequently, the mixed approach would attempt to define the smallest group of services necessary to implement APSE tools. The standard set might not allow tools to be developed requiring specialized facilities, but through controlled mutation, installations could enhance or extend the services for the purposes of efficiency or added capability. Such enhancements or extensions would have to be controlled not by the criterion of functionality, but rather by the criterion of how they are implemented. A concern in the use of nonstandard facilities in

the implementation of tools is that transportability is affected; but, by controlling the enhancements so that their use may be detected automatically by an analysis of the tool, the cost of transportability of a tool could at least be determined, if not minimized. An installation's enhancements to the set of KAPSE facilities, in the form of new routines, would be relatively easy to discover. By automatically generating a list of all services used by a tool, either directly or indirectly, and by comparing the list with the standard KAPSE-specifications, the set of nonstandard facilities could be isolated to specific sections of program text. On the other hand, however, the use of nonstandard facilities can be more difficult to detect. An example might be a tool that provides a runtime-generated nonstandard-argument to an acceptable KAPSE-service. Simple analysis of the procedure-names invokable by a tool would not reveal this type of extension, and detection would instead require a runtime monitor of KAPSE services. A further question is how the implementation of the extensions relates to the implementation of the standard KAPSE-facilities. Certainly the meaning of a standard facility could not be changed by an extension, although an extension could require that a standard service perform actions in addition to those standardized. For example, if a service called CREATE_PROCESS were part of the standardized set then it may be that an extension would require CREATE_PROCESS to store information about the process that augments what is required by the standard. The additional information may later be used by a nonstandard feature.

The mixed approach would only be applicable in the case that a standardized set of efficiently-implementable services cannot be designed as the KAPSE. Such an approach admits that transportability can be achieved only by sacrificing flexibility and/or efficiency. The advantage to this approach is that each tool may be designed and implemented according to its intended use. If the tool is to be usable on several different APSE's then it would be implemented in terms of the standard facilities only. If the tool were to resident only on one APSE then it could be implemented taking full advantage of underlying host-pecularities.

Of the alternative implementation-strategies that have been presented in this section, the evolution of a single set of standardized KAPSE-facilities, to which strict adherence is required, was the most desirable from the criteria of cost of validation and transportability. Currently however, not enough is known about the specific architecture of a machine-independent set of facilities to determine whether a single standard-KAPSE can evolve. The important question is whether a single set of services can be defined that are economically implementable on various existing systems and that are computationally efficient in terms of time and space requirements. If such a set can indeed be defined then the most desirable alternative is a single standard KAPSE whose facilities and overlying tools are validated using either the big-bang or divide-and-conquer approach. In the event that a reasonable KAPSE cannot be defined then careful attention and further work should be devoted to the mixed

approach. Questions that need to be addressed should this alternative be adopted include: What liberty can be taken in developing extensions and enhancements to the standard KAPSE? To what extent, if any, can the meanings of standard facilities be augmented? Is there a specific syntax that should be used in enhancements? What facilities should be included in the minimal standard KAPSE? Should those facilities only be sufficient to implement the MAPSE level or should they also support projected APSE tools? Without careful guidance the mixed approach could easily lead to the current situation in which vastly-different APSE structures are supported.

CONCLUSIONS AND RECOMMENDATIONS

AN APSE REFERENCE MODEL

This report has raised the issue of the need for a more substantial and extensible model for the definition of APSEs that was presented in the Stoneman requirements [2]. The model presented by Buxton was intended as being merely illustrative and limited in scope to that report and not intended to be followed closely in constructing or implementing actual support-environments. It is the recommended that the Open Systems Interconnection Model be accepted as the underlying model of APSEs and that implementations be required to clearly delineate the layers and restrict modules in which they are specified to reside.

Extensions to include Networking Environments

It is clear to us that the Ada systems are likely to exist in environments which are more extensive than considered in previous reports. That is, the Ada systems, due to their reliance on multi-processing environments and cross-system development will be installed in networks which will require a more detailed consideration of inter-system communications than has been previously presented. Such environments will need to be developed specifically for the Ada systems and should be implemented so as to permit their clarity of interconnection with Ada Programming Support Environments and, by implication, with Ada run-time environments. It is recommended that there be developed a "Strawman" to extend Ada systems into a networking environment, based on the OSI Reference Model.

The Need for Security Considerations

It is clear that Ada environments will be required to contain security elements which will provide both access security and "physical" security. Considering the OSI model, a security sublayer could be introduced into the presentation layer which would insulate the Ada programs or the APSE tools from the KAPSE layer. Other elements may need to be added at lower layers to provide security between the software and hardware systems. By insisting that the only means of access to a system is through an application layer, full security can be ensured. It is recommended that the security aspects of the design of APSEs be investigated and that the results of this study be incorporated into the Stoneman requirements.

THE NEED FOR A SINGLE KAPSE DEFINITION

In view of their potential for violating the general Ada program requirements for consistency and portability implied by Steelman and Stoneman requirements, the continued development of separate KAPSEs by the Dept. of the Army and the Air Force, it is recommended that there shall be only one KAPSE definition and

that by FY86 all interim KAPSEs be required to conform to this single model.

Defining a Standard KAPSE Completely

Following if the decision regarding the use of a single KAPSE model is followed it is recommended that work be initiated to define such a KAPSE in a form which would permit conforming implementations and ensure the adequacy of a validation procedure.

THE NEED TO DEFINE CONFORMANCE AND VALIDATION WITHIN APSE SPECIFICATIONS

The need to develop procedures and test suites to validate APSEs itself requires that some guidance be given to those responsible for the administration of those procedures. It is recommended that guidelines be established to ensure that the specifications for APSEs be accompanied by statements which specify the requirements for conformance and the conditions to be met to satisfy the validation requirements.

CONTINUED DEVELOPMENT OF FORMAL DEFINITION TECHNIQUES FOR ADA

Work is already under way, supported by AJPO, to develop a formal definition of the programming-language Ada; it is recommended that this work be extended to consider the use of a semantic-description method in connection with APSEs and specifically for the definition of conformance and validation requirements.

REFERENCES

[1] Carlson, W.E., Druffel, L.E., Fisher, D.A., and Whitaker, W.A., "Introducing Ada", Proc. 1980 ACM Ann. Conf., ACM, New York NY, 1980, 539 pp.

[2] Buxton, J.N., Requirements for Ada Programming Support Environments, "STONEMAN", U.S. Dept. of Defense, February 1980, pp.50.

[3] OSI, Information Processing Systems -- Open Systems Interconnection -- Basic Reference Model, Draft Proposal ISO/DP 7498, TC97/SC16 N890, ANSI (Secretariat), New York NY, 1982 Feb. 04, pp.78.

[4] Department of Defense, Requirements for High Order Computer Programming Languages -- "STEELMAN", Defense Advanced Research Projects Agency, Washington DC, 1978 June, pp.22.

[5] Myers, G.J., "The Art of Software Testing", John Wiley and Sons, 1979.

[6] Goodenough, J.B., and Gerhart, S.L., "Toward a Theory of Test Data Selection", IEEE Transactions on Software Engineering, (SE-1) 1975, pp. 156-173.

[7] Jackson, M.A., "Principles of Program Design", Academic Press (London), 1975.

[8] Softech Inc., "Ada Package Specifications for Ada Language System KAPSE", Working Copy, June, 1982.

[9] Memorandum of Agreement among Deputy Under Secretary (AM) Assistant Secretary of the Army (RD&A), Assistant Secretary of the Navy (RE&S), and Assistant Secretary of the Air Force (RD&L), on an APSE to be shared by the three Military Departments, January, 1982.

[10] Lyons, T., Aims of the KITIA and Alternative Approaches to the KAPSE, Private Communications, 1982.

[11] Wrege, D.E., "The Consequences of Multiple DOD KAPSE Efforts", Working Paper of the KITIA, 1982..

OFFICE OF NAVAL RESEARCH

Engineering Psychology Group

TECHNICAL REPORTS DISTRIBUTION LIST

OSD

CAPT Paul R. Chatelier
Office of the Deputy Under Secretary
of Defense
OUSDRE (E&LS)
Pentagon, Room 3D129
Washington, D. C. 20301

Dr. Dennis Leedom
Office of the Deputy Under Secretary
of Defense (C^3I)
Pentagon
Washington, D. C. 20301

Department of the Navy

Engineering Psychology Group
Office of Naval Research
Code 442 EP
Arlington, VA 22217

Communication & Computer Technology
Programs
Code 240
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Information Sciences Division
Code 433
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Special Assistant for Marine Corps
Matters
Code 100M
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Dr. J. Lester
ONR Detachment
495 Summer Street
Boston, MA 02210

Department of the Navy

Mr. R. Lawson
ONR Detachment
1030 East Green Street
Pasadena, CA 91106

CDR James Offutt, Officer-in-Charge
ONR Detachment
1030 East Green Street
Pasadena, CA 91106

Director
Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D. C. 20375

Dr. Michael Melich
Communications Sciences Division
Code 7500
Naval Research Laboratory
Washington, D. C. 20375

Dr. J. S. Lawson
Naval Electronic Systems Command
NELEX-06T
Washington, D. C. 20360

Dr. Robert E. Conley
Office of Chief of Naval Operations
Command and Control
OP-094H
Washington, D. C. 20350

Dean of Research Administration
Naval Postgraduate School
Monterey, CA 93940

CDR Jack Kramer (5 copies)
ADA Joint Program Office
801 N. Randolph Street
Ballston Tower #2, Room 1210
Arlington, VA 22703

Department of the Navy

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
Code RD-1
Washington, D. C. 20380

Dr. L. Chmura
Naval Research Laboratory
Code 7592
Computer Sciences & Systems
Washington, D. C. 20375

Chief, C^3 Division
Development Center
MCDEC
Quantico, VA 22134

Human Factors Technology Administrator
Office of Naval Technology
Code MAT 0722
800 N. Quincy Street
Arlington, VA 22217

Commander
Naval Air Systems Command
Human Factors Programs
NAVAIR 334A
Washington, D. C. 20361

Mr. John Impagliazzo
Code 101
NUSC-Newport
Newport, RI 02840

Mr. Philip Andrews
Naval Sea Systems Command
NAVSEA 03416
Washington, D. C. 20362

Commander
Naval Electronics Systems Command
Human Factors Engineering Branch
Code 81323
Washington, D. C. 20360

Larry Olmstead
Naval Surface Weapons Center
NSWC/DL
Code N-32
Dahlgren, VA 22448

Department of the Navy

Dr. Mel C. Moy
Navy Personnel Research and
Development Center
Command and Support Systems
San Diego, CA 92152

CDR J. F. Funaro
Code 602
Human Factors Engineering Division
Naval Air Development Center
Warminster, PA 18974

Human Factors Engineering Branch
Code 1226
Pacific Missile Test Center
Point Mugu, CA 93042

Dean of the Academic Departments
U.S. Naval Academy
Annapolis, MD 21402

Department of the Army

Mr. J. Barber
HQS, Department of the Army
DAPE-MBR
Washington, D. C. 20310

Dr. Edgar M. Johnson
Technical Director
U.S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Technical Director
U.S. Army Human Engineering Labs
Aberdeen Proving Ground, MD 21005

Department of the Air Force

U.S. Air Force Office of Scientific
Research
Life Sciences Directorate, NL
Bolling Air Force Base
Washington, D. C. 20332

AFHRL/LRS TDC
Attn: Susan Ewing
Wright-Patterson AFB, OH 45433

Department of the Air Force

Chief, Systems Engineering Branch
Human Engineering Division
USAF AMRL/HES
Wright-Patterson AFB, OH 45433

Dr. Earl Alluisi
Chief Scientist
AFHRL/CCN
Brooks Air Force Base, TX 78235

Other Government Agencies

Defense Technical Information Center
Cameron Station, Bldg. 5
Alexandria, VA 22314

Dr. Craig Fields
Director, System Sciences Office
Defense Advanced Research Projects Agency
1400 Wilson Blvd.
Arlington, VA 22209

Other Organizations

Dr. Deborah Boehm-Davis
General Electric Company
Information Systems Programs
1755 Jefferson Davis Highway
Arlington, VA 22202

Mr. Edward M. Connelly
Performance Measurement Associates, Inc.
410 Pine Street, S. E.
Suite 300
Vienna, VA 22180

Mr. Richard Main
ONR Resident Representative
George Washington University
2110 G. Street, N.W.
Washington, D.C. 20037

Dr. John J. O'Hare
Code 455
Office of Naval Research
800 N. Quincy Street
Arlington, VA 22217

Dr. E. Gloye
ONR Western Regional Office
1030 East Green Street
Pasadena, CA 91106

Other Organizations

CDR Norman E. Lane
Code N7A
Naval Training Equipment Center
Orlando, FL 32813

Dr. J. Hopson
HF Engineering Division
Naval Air Development Center
Warminster, PA 18974

Dr. A. Meyrowitz
Code 433
Office of Naval Research
800 N. Quincy Street
Arlington, VA 22217

Dr. Thomas McAndrew
Code 32
Naval Undersea Systems Center
New London, CT 06320

Mr. Walter P. Warner
Code KOZ
Strategic Systems Department
Naval Surface Weapons Center
Dahlgren, VA 22448

Dr. Richard Neetz
Pacific Missile Test Center
Code 1226
Pt. Mugu, CA 93042

Mr. Rick Miller
NSWC
Code N32
Dahlgren, VA 22448

Dr. Arthur Fisk
ATT Long Lines
12th Floor
229 W. Seventh St.
Cincinnati, OH 45202

Dr. Lou Chmura
Code 7592
Naval Research Laboratory
Washington, D.C. 20375

Dr. R. J. K. Jacob
Code 7590
Naval Research Laboratory
Washington, D.C. 20375